INSTITUT DE LA FRANCOPHONIE POUR L'INFORMATIQUE

TP N°1

ARCHITECTURE DES RESEAUX

CONCEPTION D'UN CLIENT FTP

Superviseur :

- M. Victor Moraru

Réaliser par: Groupe 6

- Ho vi Dai
- Levan Hung
- Nguyen Thy Anh Tuyet
- Nguyen Thi Thuy Nga
- Ewelle Ewelle RICHARD

Hanoï, Avril – 2009

Sommaire

Figures

1. INTRODUCTION

1.1.Objectif du document

Ce document constitue le rapport de la conception et de l'implémentation d'un client FTP effectué dans le cadre du cours Architecture et réseau. Pour pouvoir bien présenter notre travail, nous avons commencé par faire un léger rappel des notions fondamentales du protocole FTP tel que définit par la RFC 959, ensuite nous montrerons les détails de l'implémentation des commandes relatives au client FTP, et enfin nous ferons quelques testes et captures des trames d'une connexions afin d'apprécier les résultats et les performances du programme développé.

1.2. Spécification du sujet

De manière générale, ce TP a pour but la création d'un client FTP mettant à la disposition de ses utilisateurs des commandes de base du protocole FTP conforme au standard de FTP. Les commandes du RFC959 implémentées dans ce programme sont les suivants : USER, PASS, SYST, CWD, LIST, RETR, STOR, QUIT.

L'ensemble des commandes de ce programme sont donc les suivantes :

- **open [-s nom or ip] [-p port]** : ouvrir une connexion TCP avec le serveur
- **name** : nom_d_utilisation : envoyer l'identification de l'utilisateur au serveur
- **password** : mot_de_passe : envoyer le mot de passe de l'utilisateur au serveur
- **cd** répertoire : changer le répertoire de travail courant du serveur
- **ls** [répertoire] : demander au serveur de renvoyer le contenu du

répertoire de travail courant ou d'un répertoire spécifié du serveur

- **get** file : obtenir un fichier à partir du répertoire distant courant
- **put** file : stocker un fichier dans le répertoire distant courant
- **close** : fermer la connexion ftp mais sans quitter ftp
- **quit** : quitter ftp
- **help** : afficher les commandes supportées pour le client

1.3.Aperçu du programme

1.3.1. Architercture du programme

Notre programme est constitué de 3 fichiers :

- **ftp.c :** Fichier principale du programme. Contient la procédure main() et appelle tous les autres fichiers.
- **ftplib.h :** Fichier d'entête du programme. Contient la définition de toutes les procédures appelées.
- **ftplib.c :** Contient toutes les procédures implémentées dans ce programme.

1.3.2. Compilation du programme

Pour compiler le programme, vous devez :

- Entrer dans le répertoire contenant le code source

❖ Tapez la commande : **make ftp**

Le fichier makefile va se charger de compiler tous les fichiers et de créer les fichiers exécutables.

- ftplib.c → ftplib.o
- ftp.c → ftp.exe

1.3.3. Exécution du programme

❖ La première opération à effectuer c'est de lancer le programme.
Commande : ./**ftp**

❖ Ensuite effectuer une connexion au serveur :
Commande : **ftp**> **open** jupiter.dorsale.ifi

❖ Ensuite il vous sera demandé de vous authentifier ; faite le en entrant votre mot de passe et votre mot de passe.
Commandes : **name** : username
password: thepass

❖ Une fois la connexion établie, vous pouvez donc effectuer les autres commandes dans le prompt **ftp**>

- **ftp**> AIDE
- **ftp**> get image.jpg
- **ftp**> put file.txt
- **ftp**> ls ou **ftp**> ls répertoire
- **ftp>** close
- **ftp>** cd répertoire
- **ftp>** quit.

2. PROTOCOLE FTP (RFC 959)

Le **F**ile **T**ransfer **P**rotocol (protocole de transfert de fichiers), ou FTP, est un protocole de communication dédié à l'échange informatique de fichiers sur un réseau TCP/IP. Il permet, depuis un ordinateur, de copier des fichiers depuis ou vers un autre ordinateur du réseau, d'administrer un site web, ou encore de supprimer ou modifier des fichiers sur cet ordinateur.

2.1.FONCTIONNEMENT DE FTP

FTP obéit à un modèle client-serveur, c'est-à-dire qu'une des deux parties, le client, envoie des requêtes auxquelles réagit l'autre, appelé serveur. En pratique, le serveur est un ordinateur sur lequel fonctionne un logiciel lui-même appelé serveur FTP, qui rend publique une arborescence de fichiers similaire à un système de fichiers Unix. Pour accéder à un serveur FTP, on utilise un logiciel client FTP (possédant une interface graphique ou en ligne de commande).

2.1.1. Le modèle FTP

Lors d'une connexion FTP, deux canaux de transmission sont ouverts :

- Un canal pour les commandes (canal de contrôle)
- Un canal pour les données

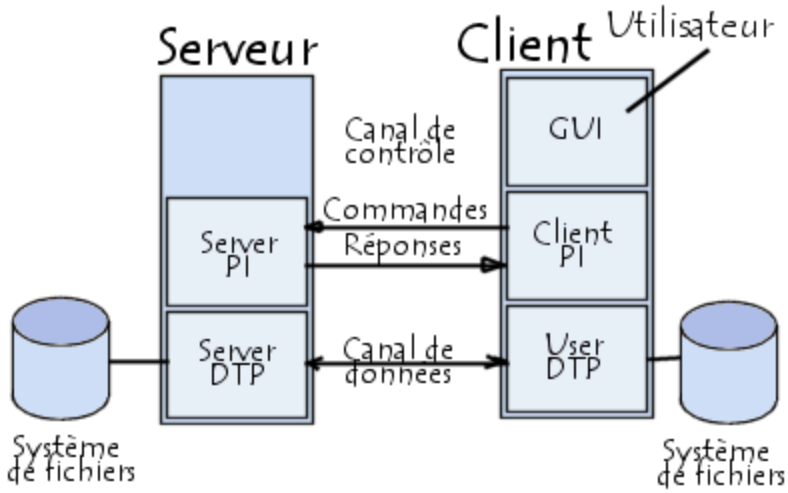

Figure 1: Modèle FTP

Le protocole, qui appartient à la couche session du modèle OSI et à la couche application du modèle ARPA, utilise une connexion TCP. Il peut s'utiliser de deux façons différentes :

- **Mode actif**: c'est le client FTP qui détermine le port de connexion à utiliser pour permettre le transfert des données.
- **Mode passif**: le serveur FTP détermine lui même le port de connexion à utiliser pour permettre le transfert des données (data connexion) et le communique au client.

Dans le cadre de ce TP, nous utiliserons le mode passif du protocole.

2.1.2. Etablissement de connexion.

Le client établi une première session TCP sur le port 21 (FTP) du serveur ("control channel"). Une fois la session établie et l'authentification FTP acceptée, on demande au serveur de se mettre en attente de session TCP grâce à la commande PASV. Alors le client peut établir une seconde session TCP sur un port dynamique vers le serveur ("data channel").

Le numéro de port dynamique est transmis du serveur vers le client suite à la commande PASV.

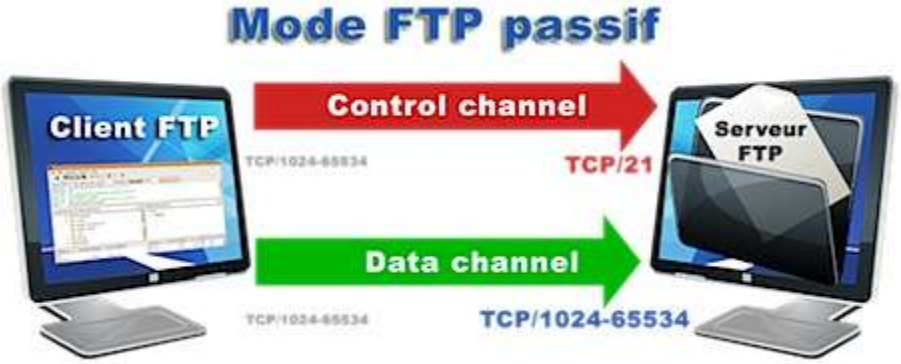

Figure 2: Mode FTP passif

2.1.3. Transfert de données

Les fichiers sont transférés uniquement par l'intermédiaire de la connexion de données. Le contrôle de connexion est utilisé pour le transfert des commandes, qui décrivent les fonctions à accomplir, et les réponses à ces commandes. Plusieurs commandes sont concernées par le transfert de données entre les hôtes. Ces commandes de transfert de données incluent :

- La commande MODE qui spécifie la façon dont les bits de données doivent être transmis.
- La commande STRUCTURE et TYPE, qui sont utilisés pour définir la manière dont les données doivent être représentés.

Dans ce TP, nous utiliserons le type binaire, La source transfère les bits du fichier à envoyer successivement les uns après les autres et le récepteur doit enregistrer les données comme une chaîne continue de bits.

3. IMPLEMENTATION DES COMMANDES

Ce paragraphe décrit l'ensemble des commandes implémentées dans ce programme, et pour chacune des commande, nous expliciterons la fonctionnalité, le fonctionnement, les échanges de flux d'informations via les deux canaux, les commandes RFC959 correspondantes et envoyées au serveur, le code de réponse, et l'algorithme d'implémentation.

- **open [nom, IP] [port]**

Cette commande nous permet d'établir une connexion TCP avec le serveur.

❖ **Processus de fonctionnement** :

- Etablir un canal de contrôle.
- Attendre la réponse 220 pour être certain que le service soit disponible pour le nouvel utilisateur.
- Identifier le nom de l'utilisateur et son mot de passe
  - Envoyer la demande RFC959 USER pour identifier le compte utilisateur
  - Envoyer la demande RFC959 PASS pour compléter les données d'indentification de l'utilisateur. Cette commande suit toujours la commande USER

❖ **Algorithme et explication** :

➢ **Algorithme**

D'abord, on faire un connexion à le serveur ftp:

```
int FtpConnect(const char *host, netbuf **nControl) {
        int sControl;
        struct sockaddr_in sin;
        struct hostent *phe;
        struct servent *pse;
        int on = 1;
        netbuf *ctrl;
        char *lhost;
        char *pnum;
```

```
        memset(&sin, 0, sizeof(sin));
        sin.sin_family = AF_INET;
        lhost = strdup(host);
        pnum = strchr(lhost, ':');
        if (pnum == NULL) {
#if defined(VMS)
                sin.sin_port = htons(21); //le port défaut 21
#else
                if ((pse = getservbyname("ftp", "tcp")) == NULL) {
                        perror("getservbyname");
                        return 0;
                }
                sin.sin_port = pse->s_port;
#endif
        } else {
                *pnum++ = '¥0';
                if (isdigit(*pnum))
                        sin.sin_port = htons(atoi(pnum));
                else {
                        pse = getservbyname(pnum, "tcp");
                        sin.sin_port = pse->s_port;
                }
        }
        if ((sin.sin_addr.s_addr = inet_addr(lhost)) == -1) {
                if ((phe = gethostbyname(lhost)) == NULL) {
                        perror("gethostbyname");
                        return 0;
                }
                memcpy((char *) &sin.sin_addr, phe->h_addr, phe->h_length);
        }
        free(lhost);
        sControl = socket(PF_INET,SOCK_STREAM, IPPROTO_TCP);
        if (sControl == -1) {
                perror("socket");
                return 0;
        }
        if (setsockopt(sControl, SOL_SOCKET,SO_REUSEADDR,
SETSOCKOPT_OPTVAL_TYPE&on,sizeof(on)) == -1)
        {
                perror("setsockopt");
                net_close(sControl);
                return 0;
        }
        if (connect(sControl, (struct sockaddr *)&sin, sizeof(sin)) == -1)
        {
                perror("connect");
                net_close(sControl);
                return 0;
        }
        ctrl = calloc(1,sizeof(netbuf));
        if (ctrl == NULL)
        {
                perror("calloc");
```

```
                net_close(sControl);
                return 0;
        }
        ctrl->buf = malloc(FTPLIB_BUFSIZ);
        if (ctrl->buf == NULL)
        {
                perror("calloc");
                net_close(sControl);
                free(ctrl);
                return 0;
        }
        ctrl->handle = sControl;
        ctrl->dir = FTPLIB_CONTROL;
        ctrl->ctrl = NULL;
        ctrl->cmode = FTPLIB_DEFMODE;
        ctrl->idlecb = NULL;
        ctrl->idletime.tv_sec = ctrl->idletime.tv_usec = 0;
        ctrl->idlearg = NULL;
        ctrl->xfered = 0;
        ctrl->xfered1 = 0;
        ctrl->cbbytes = 0;
        if (readresp('2', ctrl) == 0)
        {
                net_close(sControl);
                free(ctrl->buf);
                free(ctrl);
                return 0;
        }
        *nControl = ctrl;

        return 1;
}
```

Ensuite, après connecter le serveur, on fait le login

```
int FtpLogin(const char *user, const char *pass, netbuf *nControl) {

        char tempbuf[64];

        if (((strlen(user) + 7) > sizeof(tempbuf)) || ((strlen(pass) + 7)
                        > sizeof(tempbuf)))
                return 0;

        sprintf(tempbuf, "USER %s", user);

        if (!FtpSendCmd(tempbuf, '3', nControl)) { //on envoie la commande
                if (nControl->response[0] == '2')
                        return 1;
                return 0;
        }
        sprintf(tempbuf, "PASS %s", pass);

        return FtpSendCmd(tempbuf, '2', nControl); //on envoie la commande
```

```
}
```

- **cd <répertoire>**

Cette commande nous permet changer le répertoire de travail courant pour le sous-dossier <répertoire>.

❖ **Processus de fonctionnement** :

- ➢ Envoyer la commande RFC959 CWD au serveur
- ➢ Attendre la réponse 220 pour être certain que le service soit disponible pour

❖ **Algorithme**

```
GLOBALDEF int FtpChdir(const char *path, netbuf *nControl) {
        char buf[256];

        if ((strlen(path) + 6) > sizeof(buf))
                return 0;
        sprintf(buf, "CWD %s", path);
        if (!FtpSendCmd(buf, '2', nControl)) //on envoie la commande
                return 0;
        return 1;
}
```

- **ls**

Cette commande nous permet de demander au serveur de renvoyer le contenu de son répertoire courant.

❖ **Processus de fonctionnement** :

- ➢ Envoyer la commande RFC959 LIST pour demander au serveur de lui renvoyer le contenu d'un répertoire.

❖ **Algorithme et explication** :

- ➢ **Algorithme**

Toutes les fonctions concernantes le tranfert d'information, on appelle la fonction **FtpXfer** qui processe le fichier avec les options 'r', 'w', 'b' (si le fichier est l'image).

```
int FtpDir(const char *outputfile, const char *path, netbuf *nControl) {

        return FtpXfer(outputfile, path, nControl, FTPLIB_DIR_VERBOSE, FTPLIB_ASCII);
}

----------------------------
static int FtpXfer(const char *localfile, const char *path, netbuf *nControl,
                int typ, int mode) {

        int l, c;
        char *dbuf;
        FILE *local = NULL;
        netbuf *nData;
        int rv = 1;
```

```
        //time_t start = time(NULL);
        time_t start, end;
        time(&start);

        if (localfile != NULL) {
                char ac[4] = "w";  //les options 'w', 'r' , 'b'
                if (typ == FTPLIB_FILE_WRITE)
                        ac[0] = 'r';
                if (mode == FTPLIB_IMAGE)
                        ac[1] = 'b';
                local = fopen(localfile, ac);
                if (local == NULL) {
                        strncpy(nControl->response, strerror(errno),
                                        sizeof(nControl->response));
                        return 0;
                }
        }
        if (local == NULL)
                local = (typ == FTPLIB_FILE_WRITE) ? stdin : stdout;
        if (!FtpAccess(path, typ, mode, nControl, &nData))
                return 0;
        dbuf = malloc(FTPLIB_BUFSIZ);
        if (typ == FTPLIB_FILE_WRITE) {
                while ((l = fread(dbuf, 1, FTPLIB_BUFSIZ, local)) > 0)
                        if ((c = FtpWrite(dbuf, l, nData)) < l) {
                                printf("short write: passed %d, wrote %d¥n", l, c);
                                rv = 0;
                                break;
                        }
        } else {
                while ((l = FtpRead(dbuf, FTPLIB_BUFSIZ, nData)) > 0)
                        if (fwrite(dbuf, 1, l, local) <= 0) {
                                perror("localfile write");
                                rv = 0;
                                break;
                        }
        }
        //fprintf(stderr,"%s",nControl->response);
        time(&end);

        //double diff = difftime(time(NULL), start);
        double diff = difftime(end, start);
        diff = diff * 1000;
        printf("¥nTemps transferer: %f ms¥n", diff);

        free(dbuf);
        fflush(local);
        if (localfile != NULL)
                fclose(local);
        FtpClose(nData);
        return rv;
}
```

- **get file**

Cette commande nous permet d'obtenir un fichier à partir du répertoire distant courant.

❖ **Processus de fonctionnement** :

➢ Envoyer la commande RFC959 RETR pour demander au serveur de lui renvoyer une copie d'un ficher du serveur.

❖ **Algorithme et explication** :

➢ **Algorithme**

```
int FtpGet(const char *outputfile, const char *path, char mode,
              netbuf *nControl) {
       return FtpXfer(outputfile, path, nControl, FTPLIB_FILE_READ, mode);
}
```

- **put file**

Cette commande nous permet de stocker un fichier dans le répertoire distant courant.

❖ **Processus de fonctionnement** :

➢ Envoyer la commande RFC959 STOR pour le stockage d'un fichier local sur le serveur.

❖ **Algorithme et explication** :

➢ **Algorithme**

```
int FtpPut(const char *inputfile, const char *path, char mode,
              netbuf *nControl) {

       return FtpXfer(inputfile, path, nControl, FTPLIB_FILE_WRITE, mode);

}
```

- **close**

Cette commande nous permet d' effectuer la fermeture de la connexion sans quitter ftp.

❖ **Processus de fonctionnement** :

➢ Fermer le canal de contrôle en maquant le variable '***ouverte'*** qui signifie l'ouverture du canal de contrôle

❖ **Algorithme et explication** :

➢ **Algorithme**

```
int FtpClose(netbuf *nData) {
       netbuf *ctrl;
       switch (nData->dir) {
       case FTPLIB_WRITE:
```

```
                if (nData->buf != NULL)
                        writeline(NULL, 0, nData);
        case FTPLIB_READ:
                if (nData->buf)
                        free(nData->buf);
                shutdown(nData->handle, 2);
                net_close(nData->handle);
                ctrl = nData->ctrl;
                free(nData);
                if (ctrl) {
                        ctrl->data = NULL;
                        return (readresp('2', ctrl));
                }
                return 1;
        case FTPLIB_CONTROL:
                if (nData->data) {
                        nData->ctrl = NULL;
                        FtpClose(nData);
                }
                net_close(nData->handle);
                free(nData);

                return 0;
        }
        return 1;
}
```

- **quit**

Cette commande nous permet d'effectuer la fermeture de la connexion et/ou quitter ftp.

❖ **Processus de fonctionnement** :

➢ Envoyer la commande RFC959 QUIT au serveur et lui demander de fermer la connexion.

❖ **Algorithme et explication** :

➢ **Algorithme**

```
void FtpQuit(netbuf *nControl) {
        if (nControl->dir != FTPLIB_CONTROL)
                return;

        FtpSendCmd("QUIT", '2', nControl); //on envoie la commande

        net_close(nControl->handle);
        free(nControl->buf);
        free(nControl);
}
```

4. ILLUSTRATION DU FTP CLIENT

Cette partie présente une illustration du fonctionnement du programme afin de voir exactement comment ça marche. Pour cela, nous allons dans un premier temps proposer un scénario d'execution et ensuite nous présente les traces de l'exécution de ce scénario.

4.1. Scénario d'exécution

L'objetif ici est de tester toutes les commandes implémentée dans le cadre de ce TP.

- **Établir une connexion TCP avec le serveur jupiter.dorsale.ifi au port 21**
  - Commande : **open jupiter.dorsale.ifi**
  - Réponse : 220 – le service est disponible au nouvel utilisateur

- **Envoyer l'identificateur de l'utilisateur au serveur**
  - Commande : **name : ftpetu**
  - Réponse : 331 – l'utilisateur est valide et le mot de passe est demandé: reponse à la commande implicite **USER**

- **Envoyer au serveur le mot de passe de l'utilisateur**
  - Commande : **password : fd3Gxl**
  - Réponse : 230 – l'utilisateur est connecté et la session est ouverte: reponse à la commande implicite **PASS**

- **Afficher l'aide sur les commandes disponibles et leurs synthaxe**
  - Commande : **ftp> AIDE**

- **Demander au serveur de renvoyer le contenu de son répertoire courant**
  - Commande : **ftp> ls**
  - Réponses :
    - 200 – commande conclue : reponse à la commande implicite TYPE
    - 227 – passage le mode passif : reponse à la commande implicite PASV
    - 150 – le canal de données est ouvert : reponse à la commande implicite **LIST**

- **Obtenir un fichier à partir du répertoire distant courant.**
  - Commande : **ftp> get splash.gif**
  - Réponses :
    - 200 – commande conclue : reponse à la commande implicite TYPE
    - 227 – passage le mode passif : reponse à la commande implicite PASV
    - 150 – le canal de données est ouvert : reponse à la commande implicite **RETR**. Transfert des données

- **Stocker un fichier dans le répertoire distant courant.**
  - Commande : **ftp> put jupiter_splash.gif**
  - Réponses :
    - 200 – commande conclue : reponse à la commande implicite TYPE
    - 227 – passage le mode passif : reponse à la commande implicite PASV
    - 150 – le canal de données est ouvert : reponse à la commande implicite **STOR**. Transfert des données

- **Changer le répertoire de travail courant pour le sous-dossier rep**
  - Commande : **ftp> cd Dialang**

- **Fermeture de la connexion sans quitter ftp**
  - Commande : **close**
  - Commande implicite : **QUIT**

4.2.Traces d'exécution

Cette exécution à été effectué sur la distibution Ubuntu de Linux

- **Connexion et authentification**

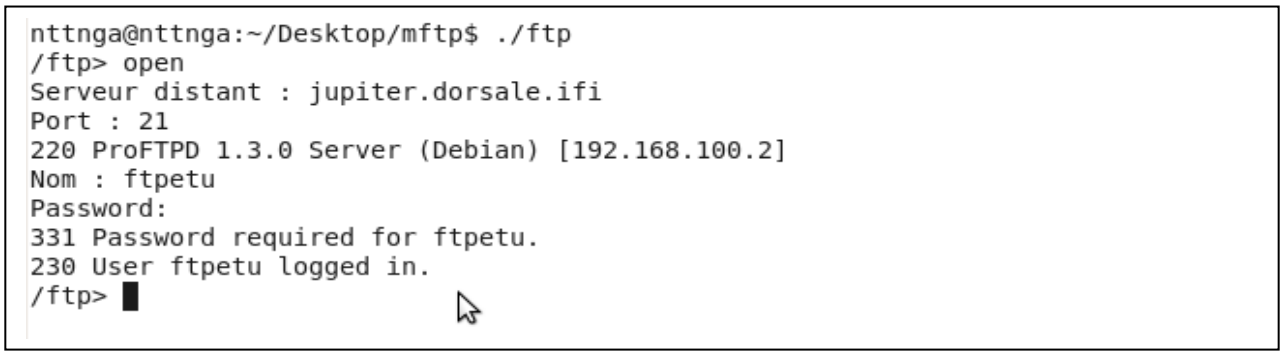

Figure 3: Connexion et authentification

- **Affichage de l'aide**

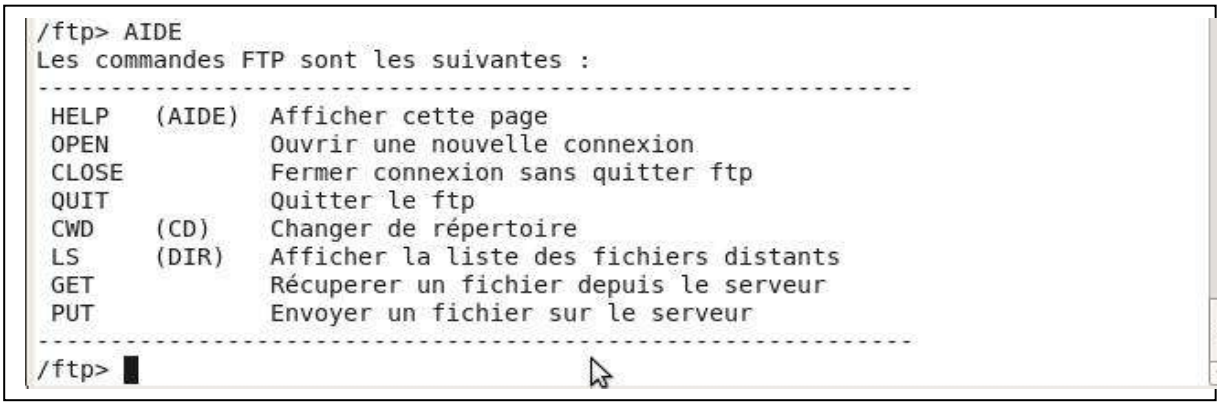

Figure 4: Affichage de l'aide

❖ Affichage du contenu du repertoire courant du serveur

```
/ftp> ls
200 Type set to A
227 Entering Passive Mode (192,168,100,2,159,78).
150 Opening ASCII mode data connection for file list
drwxrwxrwx   2 son      admin        4096 Jun 11  2008 Alphabet
drwxrwxrwx   5 echachkine prof         4096 Apr  3 06:42 B1_exemples_epreuves_pr
eparation en ligne
drwxrwxrwx   3 son      admin        4096 Jun 11  2008 B2_epreuveCIEPB2scolaire
Co
drwxrwxrwx   3 son      admin        4096 Jun 11  2008 Br?viaire d'orthographe
drwxrwxrwx  10 son      admin        4096 Mar 20 07:02 CIvilisation
drwxrwxrwx   5 son      admin        4096 Nov 24 08:24 Chansons
drwxrwxrwx   2 son      admin        4096 Jun 11  2008 Chiffres
drwxrwxrwx   2 son      admin        4096 Jun 11  2008 Compr?hension ?crite_A2
drwxrwxrwx   2 son      admin        4096 Mar 19 10:06 Compr?hension_orale
drwxrwxrwx   4 son      admin        4096 Jun 11  2008 Conjugaison
drwxrwxrwx   2 ttthoa   externes     4096 Dec  8 11:05 Court m?trage
drwxrwxrwx   6 son      admin        4096 Jun 11  2008 Diagnostic langues
```

Figure 5: Affichage du contenu du repertoire courant du serveur

❖ Telechargement de fichier

```
/ftp> get splash.gif
Fichier distant : splash.gif
Fichier local : splash_jupiter.gif
splash_jupiter.gif
200 Type set to I
227 Entering Passive Mode (192,168,100,2,136,249).
150 Opening BINARY mode data connection for splash.gif (44639 bytes)

Temps transferer: 0.000000 ms
226 Transfer complete.
/ftp>
```

Figure 6: Telechargement de fichier

❖ Upload de fichier

```
/ftp> put /home/nttnga/phdblues-ea.mp3
Fichier local : /home/nttnga/phdblues-ea.mp3
Fichier distant : phdblue
phdblue
200 Type set to I
227 Entering Passive Mode (192,168,100,2,209,134).
150 Opening BINARY mode data connection for phdblue

Temps transferer: 1000.000000 ms
226 Transfer complete.
/ftp>
```

Figure 7: Upload de fichier

❖ **Fermeture de la connexion**

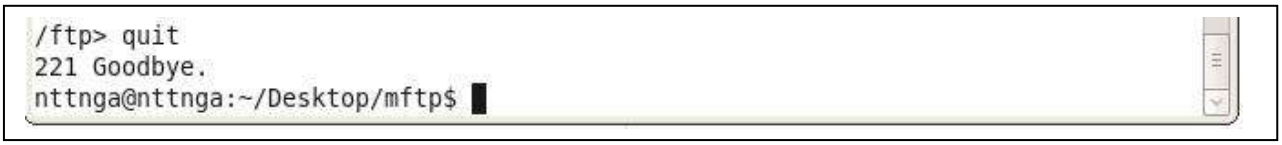

Figure 8: Fermeture de la connexion

4.3. Capture des trames et analyse des résultats

Pour prouver le fonctionnement en mode passif, ainsi que la conformité des échanges à la RFC959 de notre programme, nous allons réaliser une capture de trames entre le client et le serveur en utilisant l'outil Wireshark.

- Commande : **open jupiter.dorsale.ifi**
  Réponse : 220 – le service est disponible au nouvel utilisateur

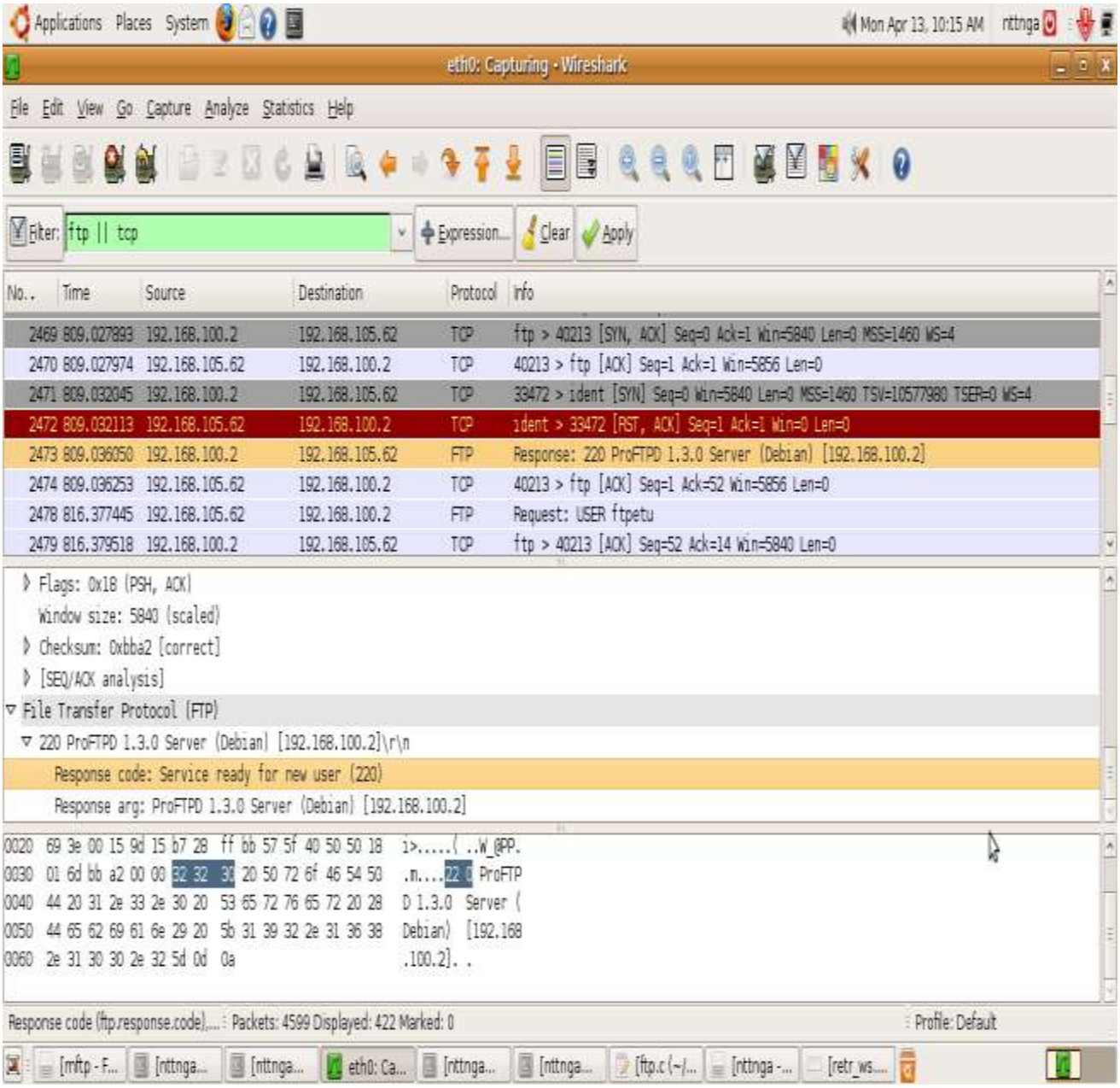

Figure 9: Capture open

Observations: Sur cette capture de trame, nous pouvons observer les évenements suivants :

- Le client initie une connexion **TCP** vers le serveur en envoyant un paquet **SYN**
- Il demande une connexion **FTP**
- Le serveur répond avec un message portant le **code 220** qui a pour signification le service est disponible. Il faut alors s'authentifier.

- Commande : **name : test**
  Réponse : 331 – l'utilisateur est valide et le mot de passe est demandé: reponse à la commande implicite **USER**

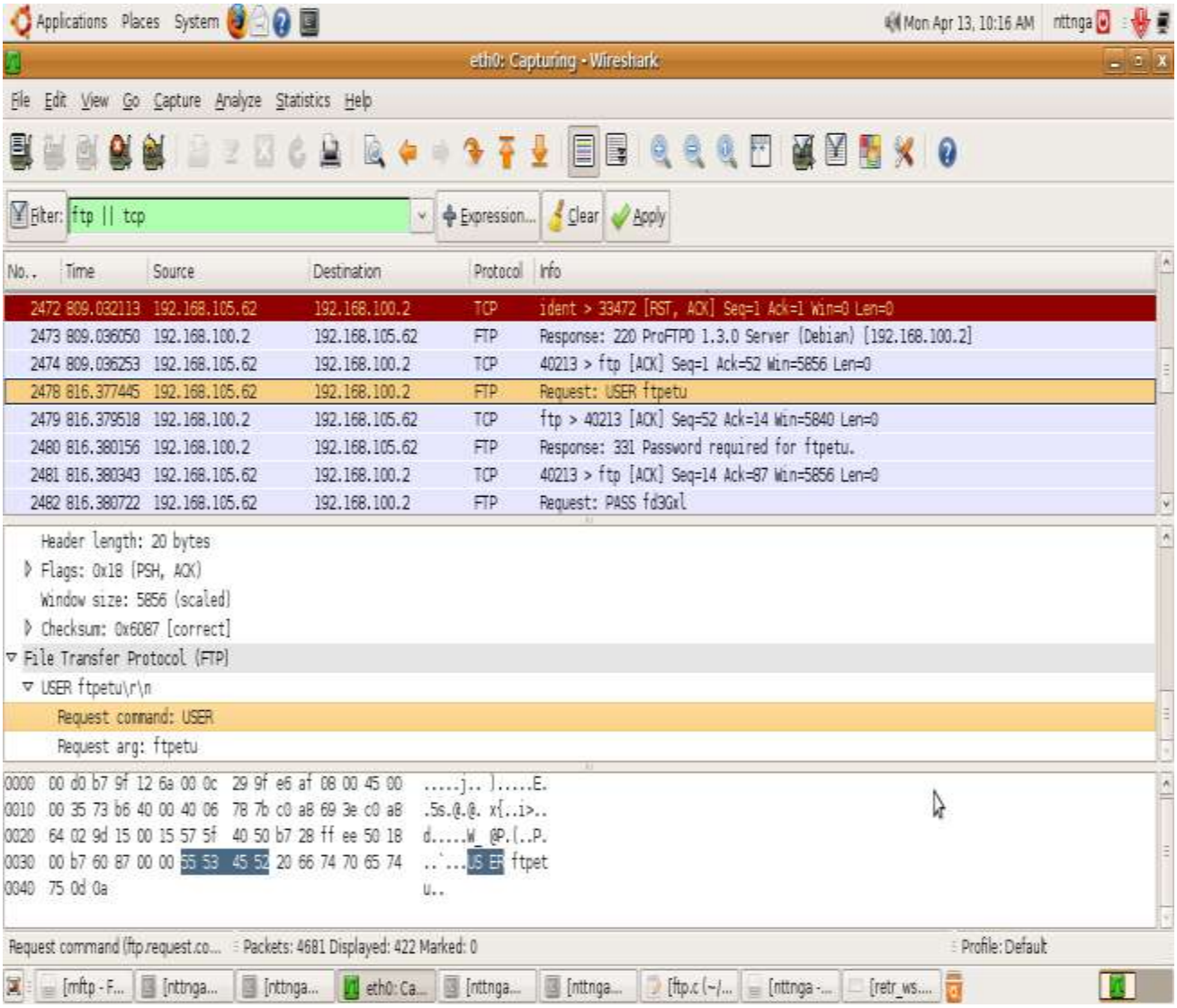

Figure 10: Capture name

Observations: Sur cette capture de trame, nous pouvons observer les évenements suivants :

- Le client envoi son nom d'utilisateur ; le système exécute l'opération en envoyant une commande **USER** vers le serveur.
- Le serveur reconnait l'utilisateur et lui demande de saisir son mot de passe avec un message ayant pour **code 331** : ce qui a pour signification l'utilisateur est valide.

- Commande : **password : v123456**
  Réponse : 230 – l'utilisateur est connecté et la session est ouverte: reponse à la commande implicite **PASS**

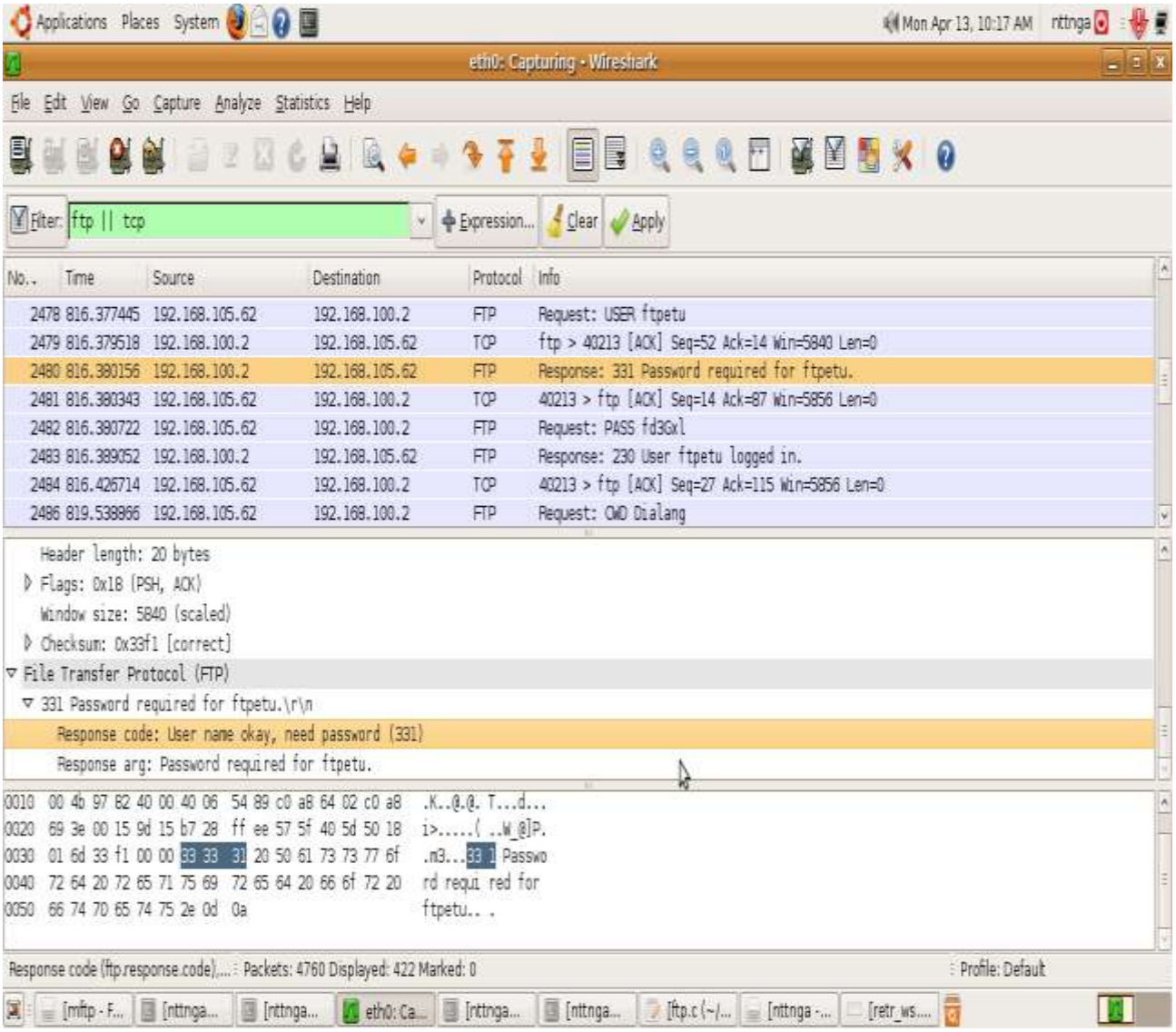

Figure 11: Capture password

Observations: Sur cette capture de trame, nous pouvons observer les évenements suivants :

- Le client saisit son mot de passe et le système exécute l'opération en envoyant une commande **PASS** vers le serveur.
- Le serveur reconnait le mot de passe comme étant celui de l'utilisateur et ouvre la connexion FTP avec le client en envoyant un message ayant pour **code 230** : ce qui a pour signification le mot de passe est valide et la connexion est ouverte.

- Commande : **ftp> ls**
  Réponses :
  - 200 – commande conclue : reponse à la commande implicite TYPE
  - 227 – passage le mode passif : reponse à la commande implicite PASV
  - 150 – le canal de données est ouvert : reponse à la commande implicite **LIST**
- Type de donnée binaire . Commande RFC : TYPE = I

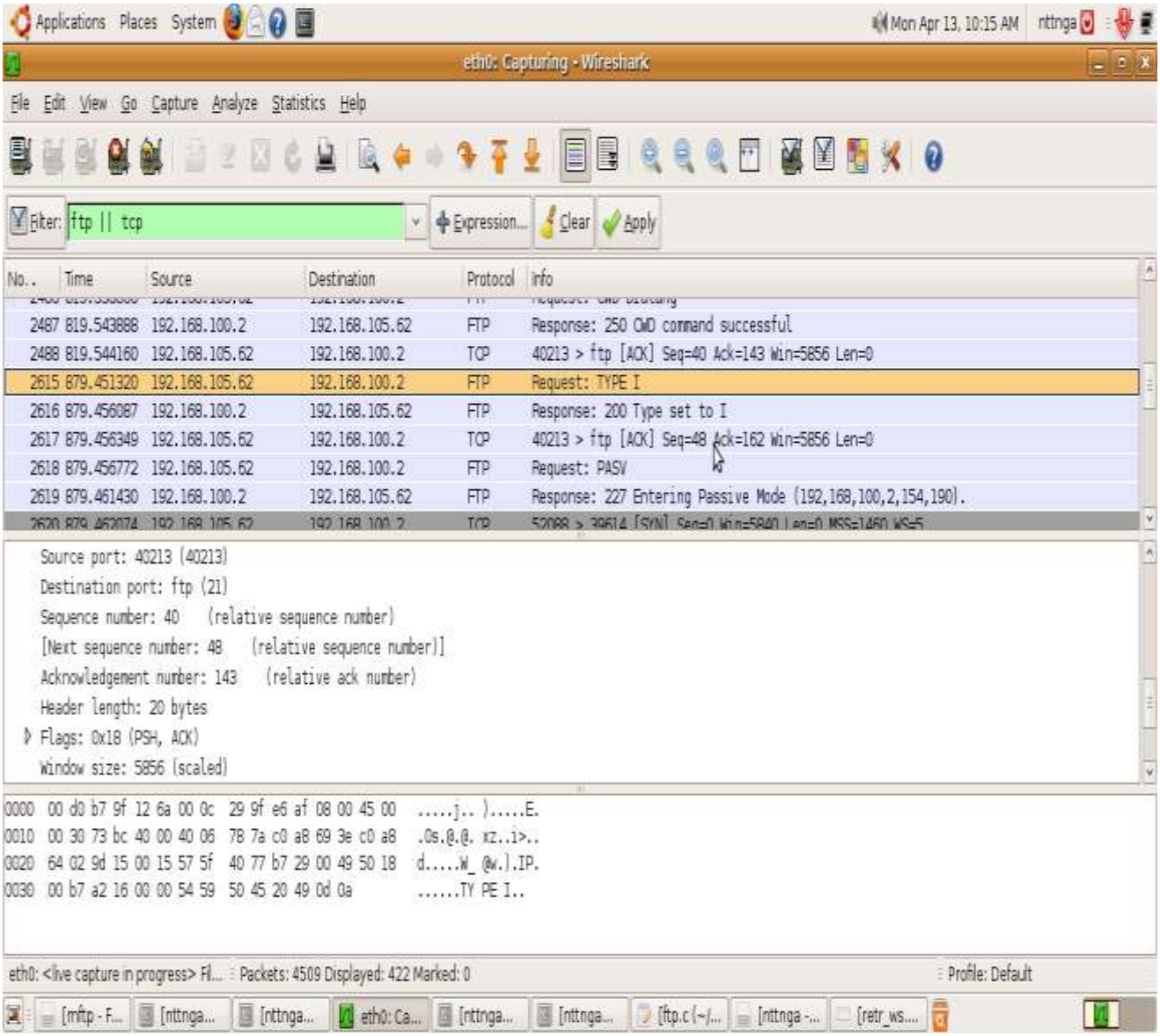

Figure 12 : Capture Type

Observations: Sur cette capture de trame, nous pouvons remarquer que les données sont transmises sur le réseau sous type binaire. TYPE = I avec pour **code 200**

- Entrée en mode passif . Commande RFC : PASV
  - Réponse : 227

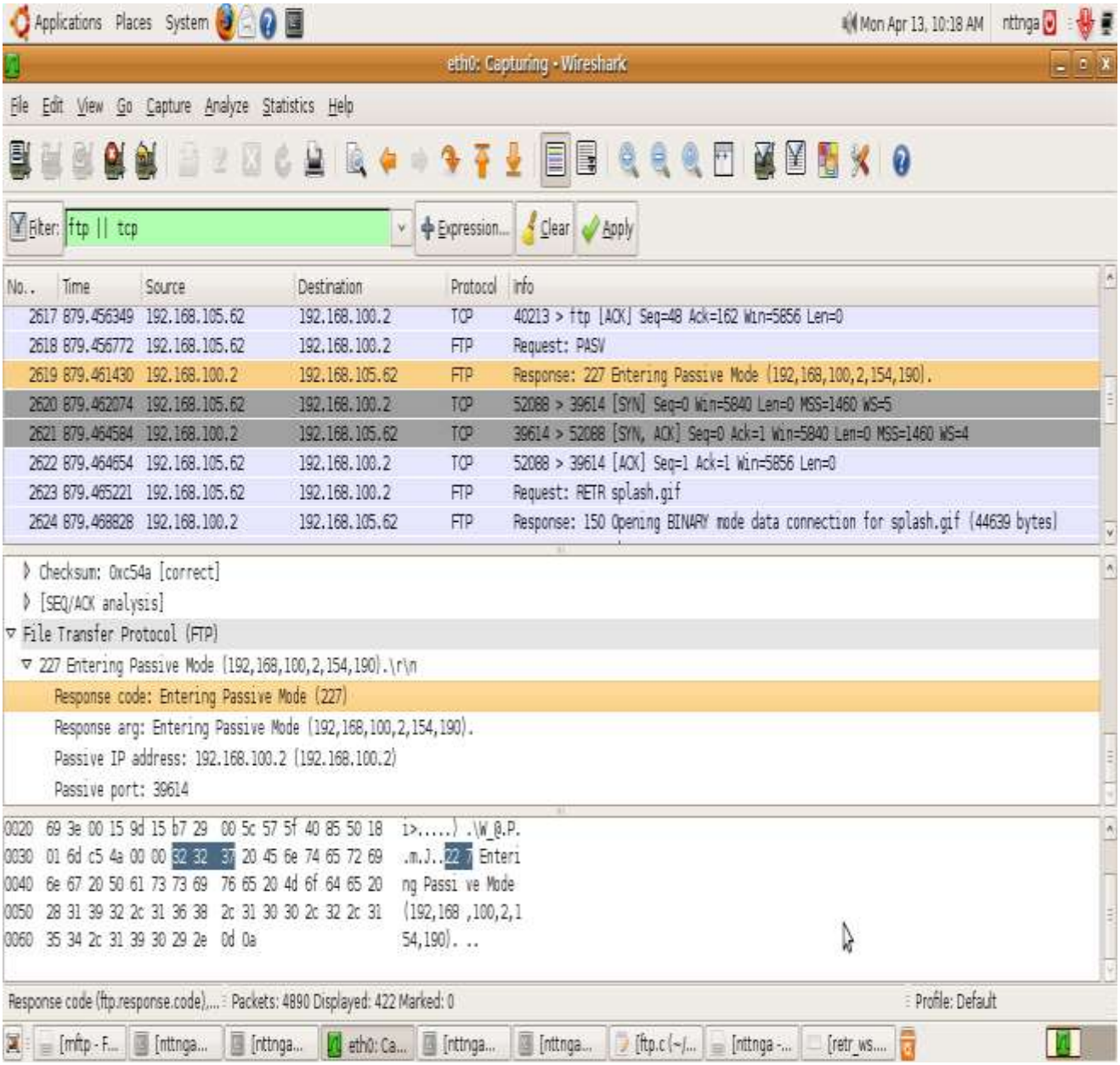

Figure 13: Capture Pasv

Observations: Sur cette capture de trame, nous pouvons remarquer que la connexion et toutes les opérations se font sur le mode passif. Avec PASV et comme réponse du serveur le **code 227**

- Lister le repertoire courant . Commande RFC : LIST
  - Réponse : 150

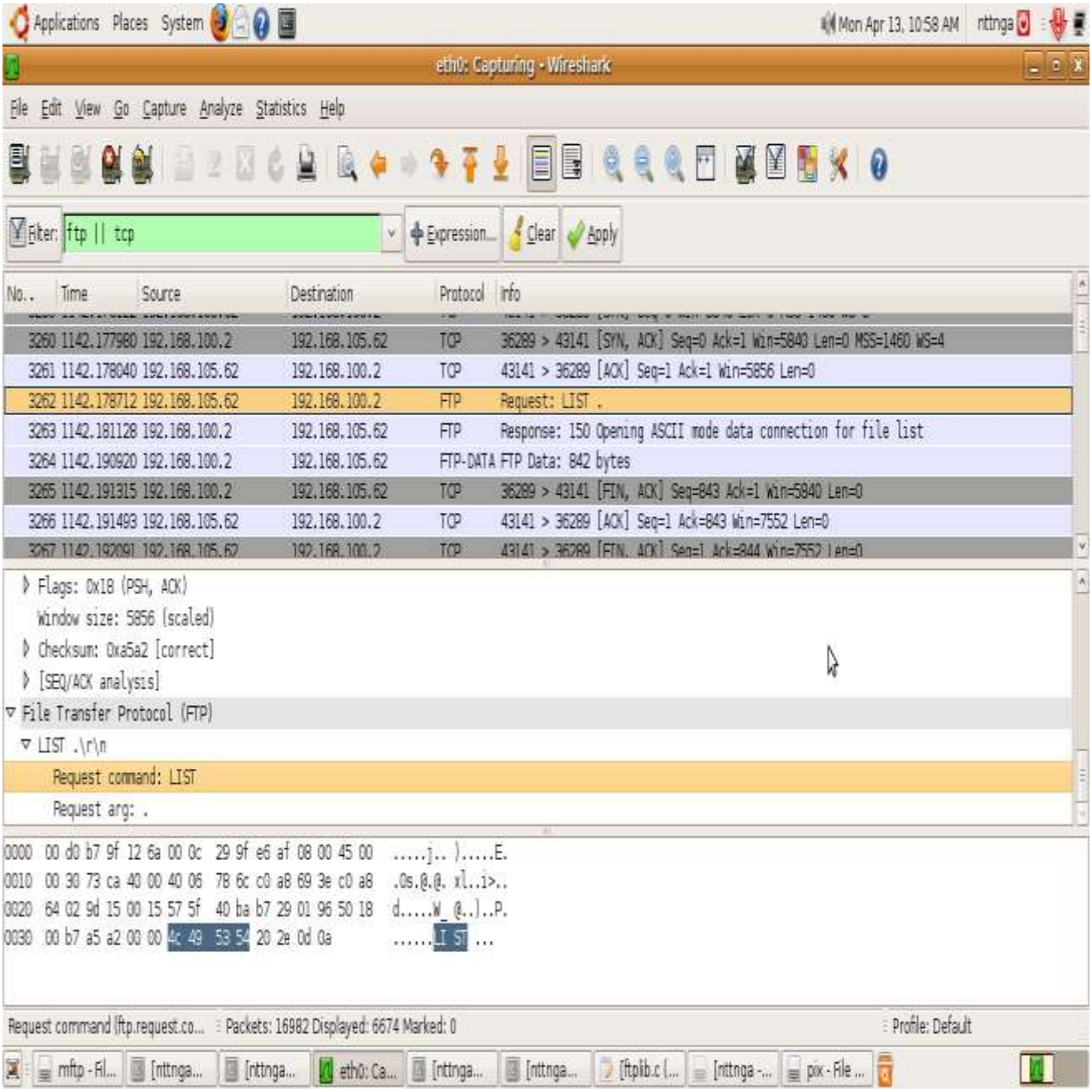

Figure 14: Capture ls

Observations: Sur cette capture de trame, nous pouvons observer les évenements suivants :

- Le client FTP envoi une demande avec la commande **LIST**
- Le serveur reçoit la demande du client et répond en lui envoyant un message avec le code 150 : ce que a pour signification d'ouvrir le canal de communication .
- Le Code **FTP-DATA** FTP data : 842 bytes : indique la taille des données transférées : données relatif a la liste des fichiers.

➢ Commande : **ftp> get image.jpg**

Réponses :

- 150 – le canal de données est ouvert : reponse à la commande implicite **RETR**. Transfert des données

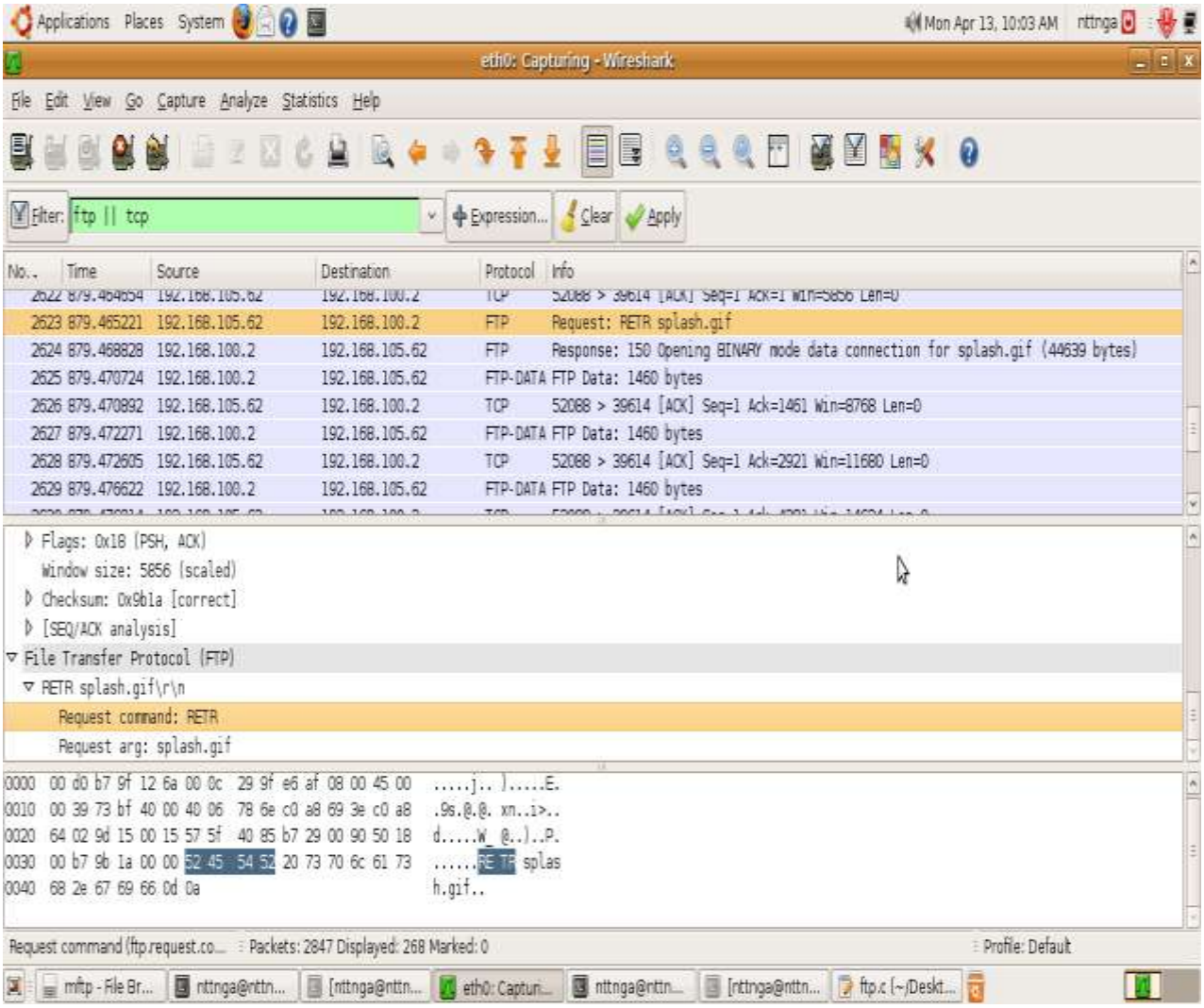

Figure 15: Capture get

Observations: Sur cette capture de trame, nous pouvons observer les évenements suivants :

- Le client FTP envoi une demande avec la commande **RETR splash.gif**
- Le serveur reçoit la demande du client et répond en lui envoyant un message avec le code 150 : ce qui ouvre le canal de communication .
- Le Code **FTP-DATA** FTP data : 1460 bytes : indique la taille des données du fichier transférées.

- Commande : **ftp> put image1.jpg**
  Réponses :
  - 150 – le canal de données est ouvert : reponse à la commande implicite **STOR**. Transfert des données

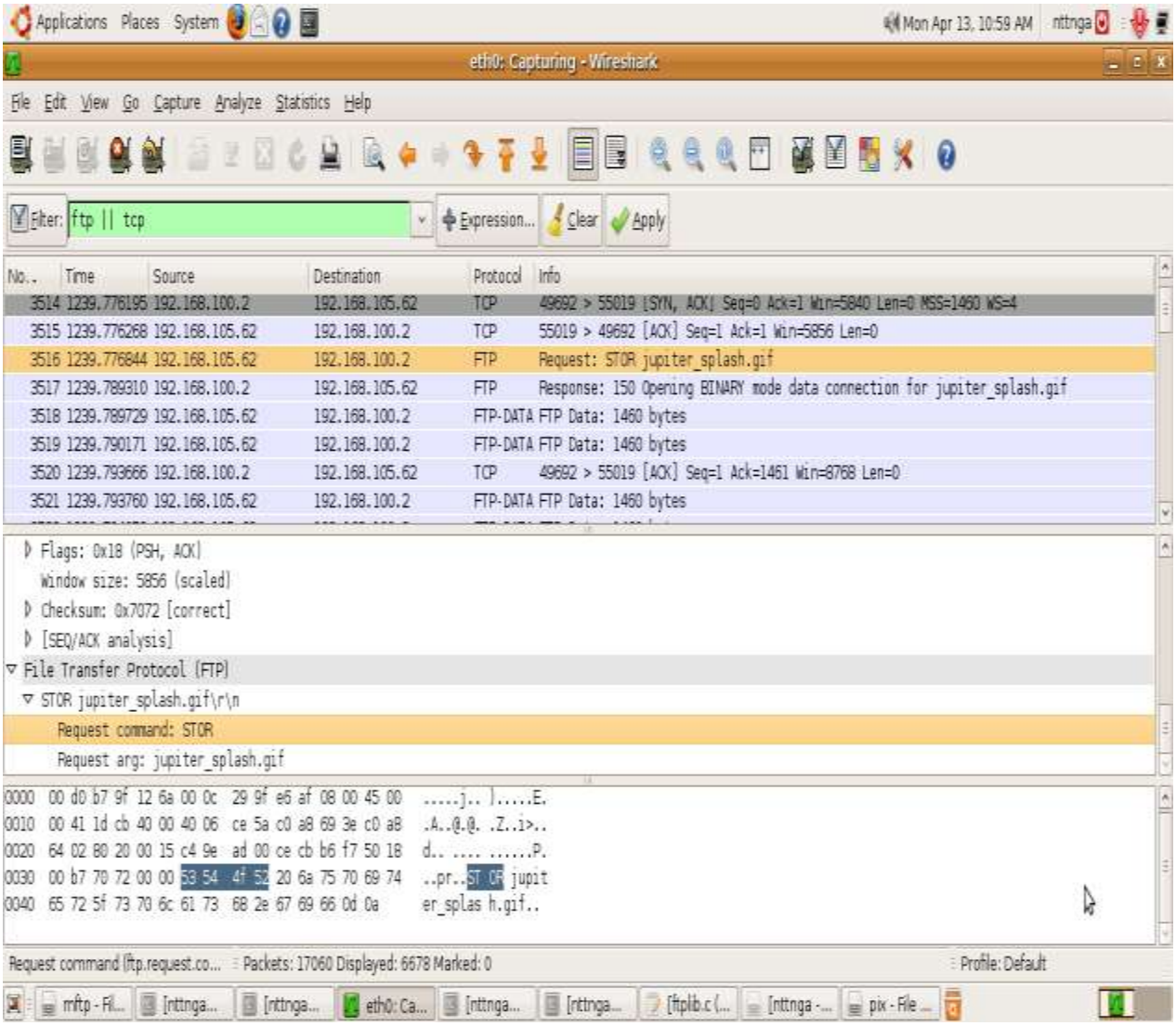

Figure 16: Capture put

Observations: Sur cette capture de trame, nous pouvons observer les évenements suivants :

- Le client FTP envoi une demande avec la commande **STOR jupiter_splash.gif**
- Le serveur reçoit la demande du client et répond en lui envoyant un message avec le **code 150** : ce qui ouvre le canal de communication .
- Le Code **FTP-DATA** FTP data : 1460 bytes : indique la taille des données transférées.

- Commande : **close**
  - Commande implicite : **QUIT**
  - Réponse 221

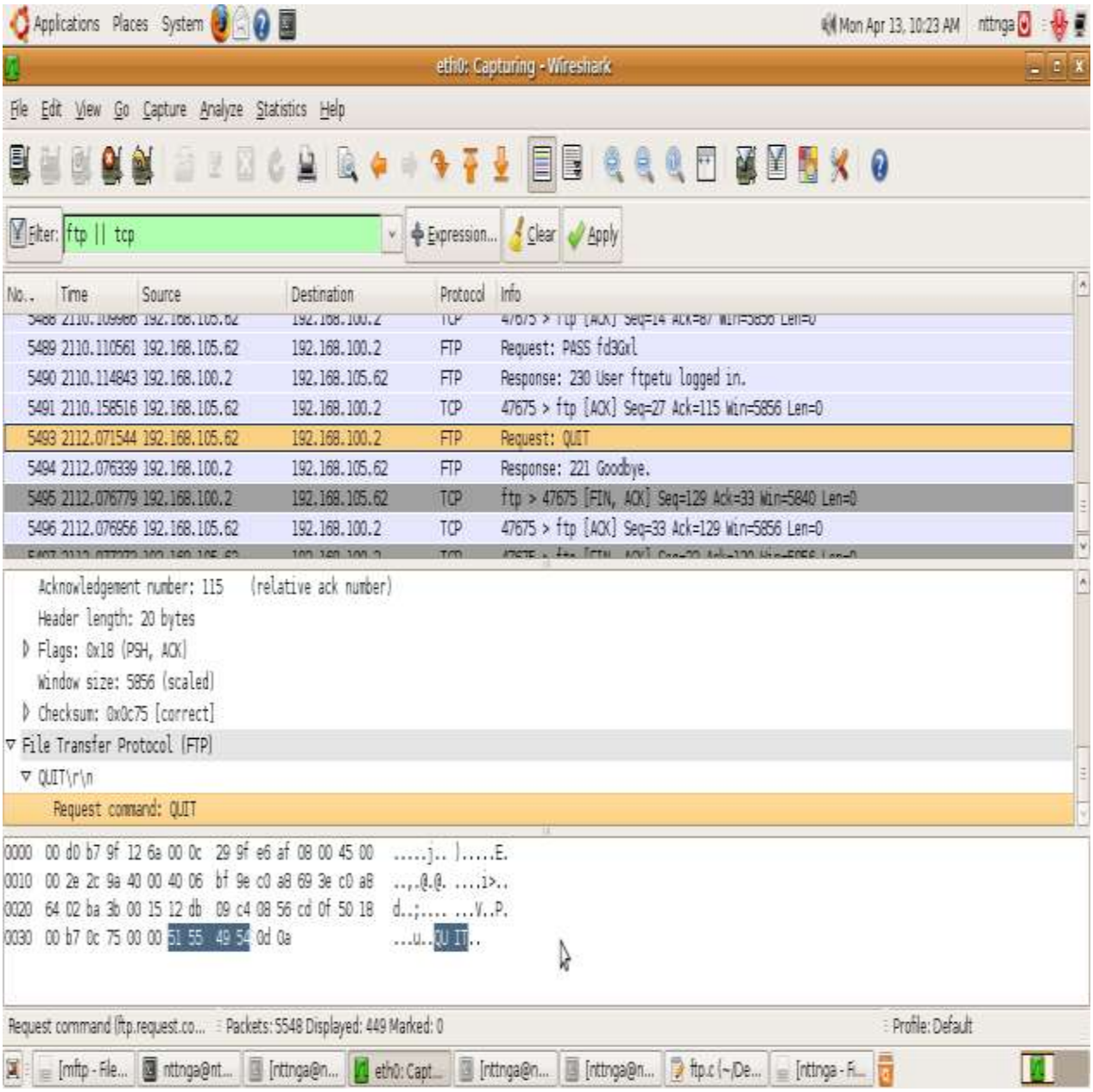

Figure 17: Capture close

Observations: Sur cette capture de trame, nous pouvons observer les évenements suivants :

- Le client FTP demande une fermeture de connexion avec la commande **QUIT**
- Le serveur reçoit la demande du client et répond en lui envoyant un message avec le **code 221** : ce qui ferme la connexion **FTP** .

5. DISCUSSION

Suite aux testes effectués, nous pouvons clairement dire que, notre programme en plus de fonctionne normalement, il satisfait toutes les spécification de ce TP. Plusque cela, nous avons eu à implémenter d'autres commandes supplémentaires et nécéssaire pour une optimisation de l'utilisation du programme : **help**, etc…

Le developpement de ce programme n'a pas été une chose très facile, nous avons été confronté à plusieurs difficultés :

- Il était parfois difficile de tester le programme avec le serveur de l'ifi car celui-ci est très instable et par moment ne permet pas l'acces et ceci même avec le FTP client d'Ubuntu.
- Il était aussi un peut difficile de trouver un serveur à intaller sur nos machines pour pouvoir tester le programme en local à la maison.
- La library que nous avons utilisée pour implémenter les fonctionnalités du programme est assez limité et le rendre conforme au standart n'était pas une chose facile.

Un inconvenient de notre application est qu'il n'y a ni sécurité, ni confidentialité ; les mots de passe et les non d'utilisateur son passé en clair dans les messages et peuvent dont être intercepté par des utilisateurs mal intensionnés.

L'interface du programme est assez simple et similaire à celui de du client FTP d'Ubuntu. Mais ceci à quelques différences près.

L'ensemble des commandes implémentées dans ce programme sont conforme aux commandes du standard, le RFC959. A chaque commande exécutée, correspond un ensemble de commande relatif au standard qui sont exécuté en background.

6. ÉVALUATION DE PERFORMANCE

Pour évaluer les performances de notre programme, nous avons effectuer quelque testes de téléchargement et de stockage de donné sur le serveur avec notre programme et également avec le client TFP d'Ubuntu et nous avons comparé les résultats en terme durée du transfert.

Fichier	Taille M octets	Notre client FTP		Client FTP d'Ubuntu	
		PUT	GET	PUT	GET
File 1	5,89	7.05 s	6.15 s	5.65 s	8,96 s
File 2	12,9	17.34 s	20.40 s	14.55 s	20,44 s
File 3	33,6	43.29 s	50.20 s	35.70 s	35,39 s

A partir de ces resultats nous constatons que les commande de notre programme et ceux d'Ubuntu ont sensiblement la même performance. Nous pouvons ainsi dire que notre application est stable et fonctionne bien.

7. PERSPECTIVES

Le client FTP que nous avons développé effectue quelques commandes de base du protocole FTP tel que définit dans le RFC959 mais il est loin d'être complet. Une perspective serait d'ameliorer le programme en y ajoutant d'autres fonctionnalités importantes comme :

- La suppression d'un fichier
- Le déplacement d'un fichier
- La modification du nom d'un fichier
- La connexion en mode actif

8. CONCLUSION

A la fin de ce travail, où il était question de concevoir et d'implémenter un client FTP dont les commandes respectent le standard RFC959, il ressort clairement que notre application remplit tout ces exigences et bien plus encore. En effet, les testes effectués et les résultats obtunus montrent que notre programme fonctionne bien et a une performance comparable à celle du client FTP d'Ubuntu. Bien que réalisant toutes les fonctionnalités essentielles d'un client FTP, ce programme pourrait t'être étendu afin d'offrir plus de commandes, par consequent plus de flexibilité aux utilisateurs.